Rust i napredni tipski sustavi

4. Rust - nastavak o upravljanju memorijom

Ivan Radiček

3. travnja 2021.

Od prošli put: vraćanje vlasništva

```
1   fn bad_length(s : String) -> usize {
2     s.len()
3   } // Nakon ovoga se memorija koja drži "s" oslobađa
4
5   fn better_length(s : String) -> (usize, String) {
6     (s.len(), s)
7   } // Nakon ovoga se memorija koja drži "s" ne oslobađa
8
9   fn main() {
10    let s1 = String::from("Ovo je testni string");
11    let l1 = bad_length(s1);
12    println!("Prva duljina: {}", l1);
13    // Ovo nije više moguće
14    // println!("{}", s1);
15
16    let s2 = String::from("Ovo je drugi testni string");
17    let (l2, s2) = better_length(s2);
18    println!("Druga duljina: {}", l2);
19    // Ovo je sad moguće
20    // println!("{}", s2);
21  }
```

Od prošli put: vraćanje vlasništva

```
1   fn bad_length(s : String) -> usize {
2     s.len()
3   } // Nakon ovoga se memorija koja drži "s" oslobađa
4
5   fn better_length(s : String) -> (usize, String) {
6     (s.len(), s)
7   } // Nakon ovoga se memorija koja drži "s" ne oslobađa
8
9   fn main() {
10    let s1 = String::from("Ovo je testni string");
11    let l1 = bad_length(s1);
12    println!("Prva duljina: {}", l1);
13    // Ovo nije više moguće
14    // println!("{}", s1);
15
16    let s2 = String::from("Ovo je drugi testni string");
17    let (l2, s2) = better_length(s2);
18    println!("Druga duljina: {}", l2);
19    // Ovo je sad moguće
20    // println!("{}", s2);
21  }
```

- Sreća, postoji bolji način na ovo . . . *posudba*!

Rust: posudba vrijednosti

- Vrijednosti se posuđuju preko reference, sa operatorom &
  - Nekad & treba navesti eksplicitno, nekad je kompajler pametan pa zna i sam
- Za neki tip `T`, referenca na `T` se označava sa `&T`
- Što se događa u memoriji sada?

```
1   fn length(s : &String) -> usize {
2     s.len()
3   } // "s" je sada samo "referenca" na string, pa se ne oslobađa
4
5   fn main() {
6     let s = String::from("Ovo je testni string");
7     let l = length(&s);
8
9     // Vlasništvo se nije promijenilo, jer je "s" samo posuđen u length
10    println!("Duljina '{}': {}", s, l);
11  }
```

Primjer automatskog &

```
1   #[derive(Debug)]
2   struct Point {
3     x: f64,
4     y: f64
5   }
6
7   impl Point {
8     fn sum(&self) -> f64 {
9       self.x + self.y // Automatski dereference (nema "->" kao npr. u C-u)
10    }
11  }
12
13  fn main() {
14    let p = Point { x: 2.2, y: 3.3};
15    let s = p.sum(); // Automatski reference (netreba eksplicitni "&")
16    // Pošto "sum" samo posuđuje "p", možemo ga i dalje koristiti
17    println!("{:?}.sum() = {}", p, s)
18  }
```

De-referenciranje pomoću *

- Obrnuti operator od & je *: vraća vrijednost iza reference/pokazivača
- Sa &T tipovima funkcionira samo ako je T : Copy
- Biti će koristan još i kasnije kada ćemo vidjeti još vrsti referenci/pokazivača

```
1  #[derive(Debug, Clone, Copy)]
2  struct Point {
3    x: f64,
4    y: f64
5  }
6
7  fn borrowd_to_owned(p : &Point) -> Point {
8    *p // Funkcionira samo zato jer je Point : Copy
9  }
10
11 fn main() {
12   let p1 = Point { x: 1.1, y: 2.2 };
13   // Isto kao: let p2 = *&p1;
14   let p2 = borrowd_to_owned(&p1);
15   println!("p1={:?}, p2={:?}", p1, p2)
16 }
```

Još malo posuđivanja

```
1  #[derive(Debug)]
2  struct Strings {
3    s1: String,
4    s2: String
5  }
6
7  fn print_strings(s : &Strings) {
8    // Moramo posuditi "s1" i "s2" (jer je i "s" posuđen)
9    let s1 = &s.s1;
10   let s2 = &s.s2;
11   println!("s1={}, s2={}", s1, s2);
12 }
13
14 fn main() {
15   let s = Strings { s1: String::from("S1"), s2: String::from("S2") };
16   print_strings(&s);
17   println!("s={:?}", s)
18 }
```

Posuđivanje s match

```
1  fn length(s : &Option<String>) -> usize {
2    match s {
3      None => 0,
4      Some(s) => s.len() // "s" je automatski posuđen ovdje
5    }
6  }
7
8  fn main() {
9    let s = Some(String::from("xyz"));
10   let l = length(&s);
11   println!("length({:?}) = {}", s, l)
12 }
```

mutable posudba

- Sa `&mut T` se označava posudba koja dozvoljava promjenu vrijednosti
- Npr. ako funkcija mijenja vrijednost argumenta (ali posuđenu, nema vlasništvo)

```
1  fn add_x(s : &mut String) {
2    s.push_str("X")
3  }
4
5  fn main() {
6    let mut s = String::from("Y");
7    add_x(&mut s);
8    println!("{}", s);
9  }
```

Različite vrste stringova u Rustu

- `String` je alociran na *heap*u i sličan je vektoru ili listi znakova (promjenjiv ili *mutable*)
- `str` je nepromjenjiv (*immutable*) niz znakova *negdje u memoriji*
  - Pošto ima nepoznatu veličinu, ne možemo ga direktno koristiti, nego samo u posuđenoj verziji `&str`
- `&str` se ponekad još zove i *slice* jer generalno govori o nekom nizu znakova (koji može biti i dio nekog drugog stringa)
- `&str` je najčešće koristi za prijenos argumenata i `&String` se automatski pretvara u njega
- `&String` je jedino koristan ako treba mijenjati vrijednost, odnosno ako se posuđuje kao `&mut String`
- Napomena: obje vrste stringova su UTF-8 stringovi

Rust : &str

- Konstatni niz znakova "..." je &str
- Vidi: https://doc.rust-lang.org/std/str/index.html i https://doc.rust-lang.org/std/primitive.str.html

```
1  fn length(s : &str) -> usize {
2    s.len()
3  }
4
5  fn main() {
6    let s = "Hello world!"; // s: &str
7    println!("length({:?}) = {}", s, length(s));
8  }
```

Posudba dijela stringa (string slice)

- Možemo posuditi i dio (slice) stringa
- Generalno o posudbi dijela nekog tipa kasnije

```
1  fn length(s : &str) -> usize {
2    s.len()
3  }
4
5  fn main() {
6    let s = "Hello world!"; // s: &str
7    // Što se događa ako ispustimo "&" ovdje?
8    let s2 = &s[0..5]; // s2: &str
9    println!("length({:?}) = {}", s2, length(s2));
10 }
```

Rust : String

- Kreira se sa `String::new()`, `String::from(&str)`, ili `s.to_string()` ako je `s : &str`
- `mut s : String` možemo i mijenjati sa `s.push_str(&str)`
- Vidi: https://doc.rust-lang.org/std/string/struct.String.html

```
1  fn main() {
2    let mut s1 = String::new(); // Novi (prazan string)
3    let s2 = "Hello world"; // &str
4    s1.push_str(s2); // Mijenjamo s1
5    let s3 = String::from(s2); // from &str (varijabla)
6    let s4 = String::from("Hello world"); // from &str (konstanta)
7    let s5 = String::from(&s4); // from &String (automatski pretvoren u &str)
8    let s6 = s2.to_string(); // Pretvorba u &str -> String
9    let s7 = &s4[0..5]; // Posudba dijela (slice) String-a
10   println!(
11     "s1={:?}, s2={:?}, s3={:?}, s4={:?}, s5={:?}, s6={:?}, s7={:?}",
12     s1, s2, s3, s4, s5, s6, s7
13   );
14 }
```

Vraćanje reference

- Ignorirati `'static` - za sada ćemo to tretirati kao i samo `&String`
- Što se događa u memoriji?

Da li je ovo OK i zašto?

```
1  fn add_x(mut s : String) -> &'static String {
2    s.push_str("X");
3    &s
4  }
5
6  fn main() {
7    let s = String::new();
8    println!("{}", add_x(s))
9  }
```

Vraćanje reference

- Ignorirati `'static` - za sada ćemo to tretirati kao i samo `&String`
- Što se događa u memoriji?

Da li je ovo OK i zašto?

```
1  fn add_x(mut s : String) -> &'static String {
2    s.push_str("X");
3    &s
4  }
5
6  fn main() {
7    let s = String::new();
8    println!("{}", add_x(s))
9  }
```

- Ne! Jer se `s` oslobađa na izlasku iz funkcije `add_x`, pa se ne može više posuditi

Rust: pravila posuđivanja

1. U bilo kojem trenutku može postojati samo **jedno** od:
   - Bilo koji broj referenci koje samo čitaju memoriju (*immutable*)
   - **Samo jedna** reference za čitanje i i pisanje (*mutable*)
2. Referenca (posudba) mora biti ispravna, odnosno pokazivati na *"živu"* memoriju

Posuđivanje - prvo pravilo

Možemo imati bilo koliko immutable posuđivanja!

```
1  fn main() {
2      let s = String::from("Hello world!");
3      let s1 = &s;
4      let s2 = &s;
5      let s3 = &s;
6      println!("s1={}, s2={}, s3={}", s1, s2, s3);
7  }
```

Posuđivanje - prvo pravilo

Ne možemo imati dva (ili više) `mutable` posuđivanja!

```
1  fn main() {
2      let mut s = String::from("Hello world!");
3      let s1 = &mut s;
4      let s2 = &mut s;
5      println!("s1={}, s2={}", s1, s2);
6  }
```

Posuđivanje - prvo pravilo

OK?

```
1  fn main() {
2    let mut s = String::from("Hello world!");
3    let s1 = &s;
4    let s2 = &mut s;
5    println!("s1={}, s2={}", s1, s2);
6  }
```

Posuđivanje - prvo pravilo

OK?

```
1  fn main() {
2    let mut s = String::from("Hello world!");
3    let s1 = &s;
4    let s2 = &mut s;
5    println!("s1={}, s2={}", s1, s2);
6  }
```

- Ne, posuđivanje `mutable` i `immutable` u isto vrijeme!

Posuđivanje - prvo pravilo

OK?

```
1  fn main() {
2      let mut s = String::from("Hello world!");
3      let s1 = &s;
4      let s2 = &s;
5      println!("s1={}, s2={}", s1, s2);
6
7      let s3 = &mut s;
8      s3.push_str("!!");
9      println!("s3={}", s3);
10 }
```

Posuđivanje - prvo pravilo

OK?

```
1  fn main() {
2      let mut s = String::from("Hello world!");
3      let s1 = &s;
4      let s2 = &s;
5      println!("s1={}, s2={}", s1, s2);
6
7      let s3 = &mut s;
8      s3.push_str("!!");
9      println!("s3={}", s3);
10 }
```

- Da, `mutable` i `immutable` posuđivanja nisu u isto vrijeme!

Posuđivanje - drugo pravilo

OK?

```
1  fn main() {
2    let n1;
3    {
4      let n2 = 42;
5      n1 = &n2;
6    }
7    println!("n1={}", n1)
8  }
```

- `n1` *živi* od linije 2-7 (nazovimo to $'a$)
- `n2` *živi* od linije 4-5 (nazovimo to $'b$)
- Problem je što želimo posuditi iz `n2` u `n1`, a $'a$ živi dulje od $'b$

Posuđivanje - drugo pravilo

OK?

```
1  fn main() {
2    let n1 = 42;
3    {
4      let n2 = &n1;
5      println!("n2={}", n2)
6    }
7    println!("n1={}", n1);
8  }
```

- `n1` *živi* od linije 2-7 (nazovimo to $'a$)
- `n2` *živi* od linije 4-5 (nazovimo to $'b$)
- Sad OK jer posuđujemo iz `n1` u `n2`, a $'a$ živi bar koliko i $'b$

Označavanje duljine života vrijednosti (`lifetime`)

- Kompajler interno svakoj vrijednosti dodijeli *lifetime* ($'a$ i $'b$ u primjeru gore)
- Nekad to nije moguće, pa trebamo ručno označiti
- Problem u funkciji niže je što kompajler ne zna koliko živi rezultat

```
1  fn longer(s1 : &str, s2: &str) -> &str {
2    if s1.len() >= s2.len() { s1 }
3    else { s2 }
4  }
5
6  fn main() {
7    let s1 = "Hello".to_string();
8    let s2 = "world!".to_string();
9    println!("Longer: {:?}", longer(&s1, &s2));
10 }
```

Eksplicitno označavanje `lifetimea`

- Slično kao kod generičkih funkcija, samo to sad nije tip `T`, nego lifetime oznaka `'a`
- Sada kompajler zna da rezultat živi isto kao i argumenti funkcije

```
1  fn longer<'a>(s1 : &'a str, s2: &'a str) -> &'a str {
2    if s1.len() >= s2.len() { s1 }
3    else { s2 }
4  }
5
6  fn main() {
7    let s1 = "Hello".to_string();
8    let s2 = "world!".to_string();
9    println!("Longer: {:?}", longer(&s1, &s2));
10 }
```

- No sada rezultat ne smije *nadživjeti* argumente

```
1  fn longer<'a>(s1 : &'a str, s2: &'a str) -> &'a str {
2    if s1.len() >= s2.len() { s1 }
3    else { s2 }
4  }
5
6  fn main() {
7    let s1 = "Hello".to_string();
8    let res;
9    {
10     let s2 = "world!".to_string();
11     res = longer(&s1, &s2);
12   }
13   println!("Longer: {:?}", res);
14 }
```

Smanjivanje života argumenta

- Kako ovo funkcionira ako `s1` i `s2` imaju različito vrijeme života a funkcije `longer` zahtjeva da imaju isto (`'a`)?
- Zato što se smijemo *praviti* da `s1` živi kraće kod poziva `longer`
- Obrnuto (da se pravimo da neka varijabla živi dulje nego što živi) nije ispravno!

```
1  fn longer<'a>(s1 : &'a str, s2: &'a str) -> &'a str {
2    if s1.len() >= s2.len() { s1 }
3    else { s2 }
4  }
5
6  fn main() {
7    let s1 = "Hello".to_string();
8    {
9      let s2 = "world!".to_string();
10     println!("Longer: {:?}", longer(&s1, &s2));
11   }
12   println!("s1={}", s1)
13 }
```

Oznaka života 'static

- Zašto ovo funkcionira?

```
1   fn longer<'a>(s1 : &'a str, s2: &'a str) -> &'a str {
2     if s1.len() >= s2.len() { s1 }
3     else { s2 }
4   }
5
6   fn main() {
7     let s1 = "Hello";
8     let res;
9     {
10      let s2 = "world!";
11      // Ovo ne funkcionira:
12      // let s2 = &("world".to_string());
13      res = longer(s1, s2);
14    }
15    println!("Longer: {:?}", res);
16  }
```

Oznaka života 'static

- Zašto ovo funkcionira?

```
1   fn longer<'a>(s1 : &'a str, s2: &'a str) -> &'a str {
2     if s1.len() >= s2.len() { s1 }
3     else { s2 }
4   }
5
6   fn main() {
7     let s1 = "Hello";
8     let res;
9     {
10      let s2 = "world!";
11      // Ovo ne funkcionira:
12      // let s2 = &("world".to_string());
13      res = longer(s1, s2);
14    }
15    println!("Longer: {:?}", res);
16  }
```

- Zato jer je svaka konstanta zapisana u statički dio memorije (nije ni stack, ni heap) koji živi cijeli program
- Tako i svaka konstanta ima u stvari tip `&'static str`, gdje `'static` označuje život cijelog programa

Oznaka života kod tipa podataka

```
1   #[derive(Debug)]
2   struct Person<'a> {
3     id: usize,
4     name: &'a str
5   }
6
7   impl<'a> Person<'a> {
8     fn new(id : usize, name: &'a str) -> Person {
9       Person { id, name }
10    }
11  }
```

Život strukture

- Naravno, definicija gore znači da vrijednost strukture `Person` ne može nadživjeti string iz kojeg je posuđeno ime
- Isto tako, primjer dolje funkcionira ako posudimo konstantu (pošto ona živi kroz cijeli program)

```
1   fn main() {
2     let p;
3     {
4       let name = "Perica".to_string();
5       p = Person::new(42, &name);
6       // OK
7       println!("{:?}", p)
8     }
9     // Nije OK
10    println!("{:?}", p)
11  }
```